cofan Documentation

Release 0.0.3

Mohammad Alghafli

Jan 30, 2019

Contents

This tutorial gives an introduction to how to use cofan python library and its features.

This is not a python tutorial. You are expected to have general knowledge in python before you start this tutorial.

CHAPTER 1

About cofan

cofan is an http server library. It is similar to python standard *http.server* libarary but with the following features:

- Serve the content of a local directory as a file browser with icons for directories and files based on their extension.
- Serve the content of a local zip file the same way as the local directories.
- List the content of a local directory in json form.
- Serve local html files as a web site.
- Organize your urls in prefix trees.
- Response differently for different ip addresses

To put it short, look at the following screenshot comparison of *http.server* and *cofan*.

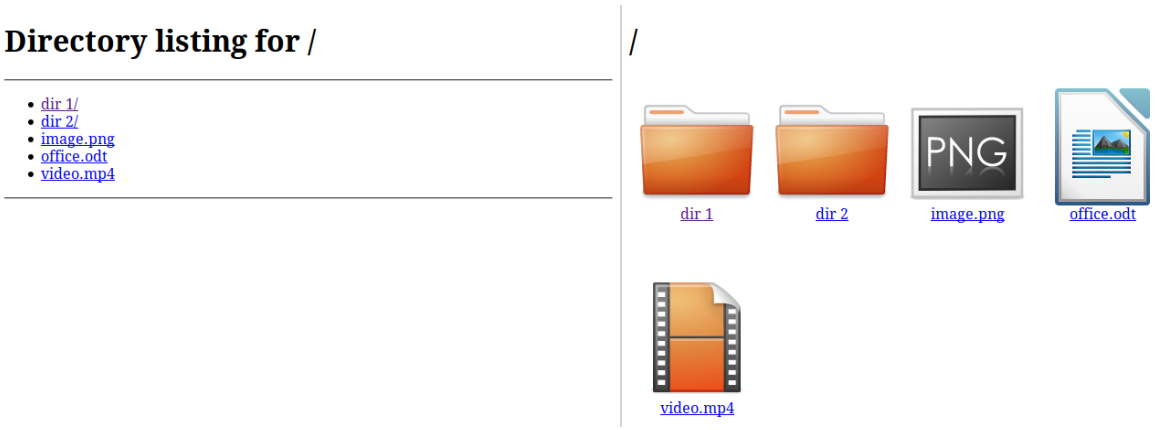

Fig. 1: Comparison of *http.server* web page (left) and *cofan* (right)

CHAPTER 2

Content

2.1 Installation

2.1.1 Requirements

The major requirement of cofan is python 3. cofan is a python 3 library and was never tested in python 2. So first make sure your version of python is 3.

Furthermore, cofan requires fileslice python library. It will be installed automatically if you use pip to install cofan.

2.1.2 Installation

Make sure you have pip for python 3 installed.

On windows install using pip by running the command:

```
pip install cofan
```

Or on linux:

```
pip3 install cofan
```

Of course, pip command should be in your PATH environment variable. If you are using windows there is a good chance pip is not in your PATH. In this case you should specify the full pip path. Search about how to use pip on windows if you are having trouble.

Try to import cofan to be sure it was installed successfully:

```
>>> import cofan
>>>
```

If it is imported without errors, you are ready to use it. You may want to have a look at one or more of the following documents:

- *Quick Guide* For those who want short and quick highlights and usage example.
- *Serving a Local Directory* This is the start of the library tutorial. It shows different library features.
- *cofan Reference* This is the library reference. All classes and functions documentation is here.

2.2 Quick Guide

2.2.1 Serve local directory

This is a typical usage example:

```
from cofan import *
import pathlib

#assume we want to share files from `~/Videos` directory

#lets make an http file browser and share our videos

#first, we specify the icons used in the file browser
#you can omit the theme. it defaults to `humanity`. This theme is supplied
#with `cofan`.
icons = Iconer(theme='humanity')

#now we create a Filer and specify the path we want to serve
vid = Filer(pathlib.Path.home() / 'Videos', iconer=icons)

#now we need to give prefixes to our website
#we create a patterner
patterns = Patterner()

#then we add the iconer and filer with their prefixes

#first, we need to add our iconer
#we should have told the iconer about its prefix but we did not. by default
#it assumes `__icons__`
#make sure to add a trailing slash
patterns.add('__icons__/', icons)

#now we add our filer as the root url
patterns.add('', vid)

#now we create our handler like in http.server. we give it our patterner
handler = BaseHandler(patterns)

#and create our server like in http.server
srv = CofanServer(('localhost', 8000), handler)

#and serve forever
srv.serve_forever()

#now try to open your browser on http://localhost:8000/
```

2.2.2 As a main python script

This module can also be run as a main python script to serve files from a directory.

commandline syntax:

```
python -m cofan.py [-a <addr>] [<root>]
```

options:

- -a <addr>, –addr <addr>: specify the address and port to bind to. <addr>

should be in the form *<ip>:<port>* where *<ip>* is the ip address and *<port>* is the tcp port. defaults to *localhost:8000*.

args:

- root: The root directory to serve. Defaults to the current directory.

2.2.3 Further readings

In *Serving a Local Directory* you can a the more detailed cofan tutorial. In *cofan Reference* you will find the library reference.

2.3 Serving a Local Directory

In this tutorial, we will look at different *cofan* features. We will start by serving files from a local directory but then we will do more complicated things like serving from a zip file and serving a website.

This tutorial is a practical tutorial. From this lesson we will write an example program and explain how it is written. Our program will start simple then will evolve with each lesson.

In this lesson, we will serve local files.

2.3.1 The beginning

First, you obviously need to install *cofan*. To start using the library, you obviously need to import it:

```
from cofan import *
```

2.3.2 The Filer class

Filer is the class used to serve local files. Before serving files, we need to make a *Filer* instance and tell it what directory to serve. Here, we will serve our videos. Our videos directory is '/home/user/Videos/'. Our program will become:

```
from cofan import *

video = Filer('/home/user/Videos/')
```

We did not start our server yet. We only made our program know what files to serve. There are still a few steps before we start serving.

2.3.3 Handling requests

We need something to handle our requests. If you have used *http.server* standard python library, cofan uses similar way with a little difference.

We need to create a *BaseHandler* object that will serve requests and we need to tell the handler that it should send incoming requests to our filer. We do this by putting our filer as an argument to the handler constructor:

```
handler = BaseHandler(video)
```

We need a few little steps to start serving files.

2.3.4 Starting the server

Similar to *http.server*, we need to create a server instance and give it the address we want to serve at and the handler to send requests to. In *cofan*, you can use the *Server* class. You can also use *http.server.Server* class if you want but unlike *http.server.Server*, cofan *Server* class can serve multiple requests at the same time.

Now we will create the server and tell it to serve at localhost:8000 and give it our handler:

```
server = cofan.Server(('localhost', 8000), handler)
```

And finally, we start serving forever:

```
server.serve_forever()
```

Our final program now becomes:

```
from cofan import *

video = Filer('/home/user/Videos/')

handler = BaseHandler(video)

server = cofan.Server(('localhost', 8000), handler)

server.serve_forever()
```

Now try to open *localhost:8000* in your web browser and you will see all your files and folders inside *Videos*. However, there are no icons yet. In fact, everything looks ugly so far. No worries as we will fix that soon.

2.3.5 Next

In next lessons, we will show file and directory icons in our website.

2.4 Showing Icons

In the previous lesson, we started serving our videos in a file browser-like webpage. However, there were no icons in our page. In this lesson, we will add icons to the page.

2.4.1 Content providers

Before we make our icons, let's take sometime learning what content providers are in *cofan*.

cofan works with the concept of content providers. A provider is an object that takes the requested URL and gives back an HTTP response code the content to be sent to the client. An example of a content provider is the *Filer* object we created in the previous lesson.

In cofan, we make a handler object and give it a single provider. Whenever the handler gets a request, it asks the provider for the content. In the previous lesson, we made a handler and told it to use our *Filer* object as its provider. We could have used a different provider if we needed and that is what we are going to do in this lesson.

2.4.2 The Iconer class

Iconer is a class used to serve icons. We create an *Iconer* object and tell our filer to use it to get the icons.

To use icons, first we need to create an *Iconer* object and tell our filer to use it:

```
iconer = Iconer()
video = Filer('/home/user/Videos/', iconer=iconer)
```

If you run the program now, you will notice nothing have changed. We still see an ugly page with no icons.

The iconer is another content provider. It is like the filer but instead of listing directory content, it serves an icon if it is asked to. In our example above, all requests go to the filer object. No request ever goes to the iconer.

We need to send some requests to the filer and some requests to the iconer. This is what we will do in the next few sections.

2.4.3 The Patterner class

We want to arrange our website to serve two things:

- Our video files.
- File icons to show them in the file browser web page.

We actually created the two providers for this, the filer and the iconer. However, we can only add a single provider to the handler.

In such situation, the *Patterner* is our friend. The *Patterner* provider is a provider that checks the requested url. Based on the url, it forwards the request to other providers.

In our example, we will make a url plan:

- If the url starts with *__icons__/* (notice the trailing slash /), we will forward the request to the iconer. In the next section, we will explain why we chose the prefix *__icons__*.
- Otherwise, we will forward the request to the video filer.

The first step to do this is to create a *Patterner* instance:

```
patterner = Patterner()
```

Then we add our iconer and filer, each with its prefix, to the patterner:

```
patterner.add('__icons__/', iconer)
patterner.add('', video)
```

Now, whenever the patterner gets a requested url which starts with '__icons__/', it will forward it to the iconer and will remove the prefix from the url. If the url does not start with *__icons__/*, it will check the other prefix. Since the other prefix is an empty string, the check will always succeed (because all strings start with an empty string).

Finally, instead of telling our handler to send requests to the filer directly, we tell the handler to forward the request to the patterner. The patterner then will forward to the other providers:

```
handler = BaseHandler(patterner)
```

Our program now becomes:

```
from cofan import *

patterner = Patterner()
iconer = Iconer()
video = Filer('/home/user/Videos/', iconer=iconer)

#add prefixes
patterner.add('__icons__/', iconer)
patterner.add('', video)

#make the handler use our patterner as its provider
handler = BaseHandler(patterner)

server = cofan.Server(('localhost', 8000), handler)

server.serve_forever()
```

Now our web site shows icons and looks better. In the next section, we will learn more about how to customize our iconer.

2.4.4 Iconer prefix

Our iconer now works and serves icons. However, how does the *__icons__/* prefix work? If you make any other prefix, it will not work.

In order to make the iconer work correctly with the filer, the iconer needs to know what prefix you will give it. The same prefix has to be given to the iconer and the patterner. Even though we did not tell the iconer what prefix we will give it, it uses the prefix *__icons__/* by default. We could have changed that to, for example, *myicons/* if we wanted. We do this by using the *prefix* argument in the *Iconer* constructor. We just need to make sure that the prefix we give to the iconer is the same prefix we add to the patterner:

```
patterner = Patterner()
iconer = Iconer()
video = Filer('/home/user/Videos/', iconer=iconer)

#add prefixes
patterner.add('myicons/', iconer)
patterner.add('', videos)
```

2.4.5 Iconer images

Another thing we can customize in our iconer is the icons to show. We can do that in two ways.

The first way is to specify an icon *theme* name. There are 3 themes that come with *cofan*: *humanity*, *plane* and *zafiro*. If no theme is specified, the iconer chooses the default theme which is *humanity*. We can change the theme using the *theme* argument in the *Iconer* constructor:

```
iconer = Iconer(theme='plane', prefix='myicons/')
```

Now our icons have changed to use the *plane* theme.

The second way to use icons is to specify a path to a zip file that contains icons using the *root* argument. For example:

```
iconer = Iconer(root='/home/user/icons.zip', prefix='myicons/')
```

In order to use a zip file as an icon theme, it should contain image files in its toplevel directory. The images can be in any format and with any extension as long as the names follow the following rules:

1- The icon for a specific file extension should be name by extension name. For example, *mp3* files icon can be names *mp3.png*, *mp3.jpg* or *mp3.<any extension>*.

#- The icon for a general file mimetype should be named with the general mimetype. For example, an icon for video files generally can be named *video.png*, *video.jpg* or *video.<any extension>*. If you do not know what a mimetype is, search for it and read about what it is.

#- The word *directory*. The icon with this name will be used as the icon for directories. For example, an icon for direcotries can be named *direcotry.png*, *direcotry.jpg* or *direcotry.<any extension>*.

#- The word *generic*. The icon with this name will be used as a fallback icon for file types that have no icons in the zip file based on rules 1 and 2 above. For example, a fallback icon can be named *generic.png*, *generic.jpg* or *generic.<any extension>*.

2.4.6 Our program so far

Below is our program so far. I have ignored the modifications we did in the last two sections:

```
from cofan import *

patterner = Patterner()
iconer = Iconer()
video = Filer('/home/user/Videos/', iconer=iconer)

#add prefixes
patterner.add('__icons__/', iconer)
patterner.add('', video)

#make the handler use our patterner as its provider
handler = BaseHandler(patterner)

server = cofan.Server(('localhost', 8000), handler)

server.serve_forever()
```

2.4.7 Next

In the next lesson, we will learn how to serve more than one local directory.

2.5 Serving Multiple Directories

In the previous lesson, we added icons to our web site. However, we were only serving our videos. In this lesson, we will serve the content of our music folder too.

2.5.1 Another Filer

The first step to serve our music is to create another *Filer*. This is easy for us because we did it in :doc: *serve-local-dir* lesson. The only difference here is the directory we are serving.

music = Filer('/home/user/Music/', iconer=iconer)

Notice that we have used the same iconer to get the icons. This way, both the video filer and the music filer will use the same icon theme.

2.5.2 Another pattern

Remember that we use the *Patterner* to forward requests to one of multiple providers based on the url prefix. our *music* is just another provider. Before we add it to the *patterner*, let make a few modifications to our previous program. We will make another url prefix plan:

- *video/* will be the prefix for the video filer.
- *music/* will be the prefix for the music filer.
- *__icons__/* will be the prefix for the iconer.

Notice that we changed the prefix for the video filer. Before, it was an empty string. Now, it is *video/*. Let's add the providers to our *pattener* now:

```
patterner.add('__icons__/', iconer)
patterner.add('video/', video)
patterner.add('music/', music)
```

We are done. Our program now becomes:

```
from cofan import *

patterner = Patterner()
iconer = Iconer()
video = Filer('/home/user/Videos/', iconer=iconer)
music = Filer('/home/user/Music/', iconer=iconer)

#add prefixes
patterner.add('__icons__/', iconer)
patterner.add('video/', video)
patterner.add('music/', music)

#make the handler use our patterner as its provider
handler = BaseHandler(patterner)

server = cofan.Server(('localhost', 8000), handler)

server.serve_forever()
```

Notice that we do not have any provider for the home page url so if we type in the browser address bar *localhost:8000*, we get a *NOT FOUND* error. We need to type *localhost:8000/video/* or *localhost:8000/music/* to open an existing page. That is not so convenient. All websites have a home page right? We are going to fix this in the next lesson.

2.5.3 A last word about prefixes

The prefixes we use in our patterner class can be any regular expression. as mentioned in a previous lesson, any trailing slash is required except for the root url. The pattern *video* will make the patterner function improperly. Use *video/* with a trailing slash instead.

Whenever a request is recieved, the patterner will do the following:

- It removes the root address and the root address trailing slash. In our example, when the requested address is *localhost:8000/video/example.mp4*, the patterner will remove *localhost:8000/*. The url now becomes *video/example.mp4*.
- Match each pattern with the beginning of the url. In our example, the url from the previous step is *video/example.mp4*. The patterner will look for the prefix that the url starts with. The prefix in this case will be *video/*.
- Remove the prefix from the url. Now the url becomes only *example.mp4*.
- Forwards the request and the new url to the corresponding provider.

Now the video filer will get a request with the url *example.mp4* only and will search for this file in its root directory.

2.5.4 Next

In the next lesson, we will make a home page for our website. Our home page will have links to our video page and music page.

2.6 Listing Prefixes

In the previous lesson, we made a music page in addition to the video page. However, there is nothing in the home page of our website. In this lesson, we will make a home page for our website which will contain a link to our video page and another link for our music page.

2.6.1 Pattern lister

We need the *PatternLister* provider for this job. The *PatternLister* is used to create a page with a list of all the prefixes in a *Patterner* object so that users can click on an item and open the corresponding page. This will help us to create our home page.

First, we need to create a *PatternLister* object and tell it to list prefixes from our patterner:

```
lister = PatternLister(patterner)
```

In our example, There are three prefixes in the *pattener*: *video/*, *music/* and *__icons__/*. Our *lister* now knows them.

Remember that the *pattener* is the first provider that handles a request and it will forward it to other providers. In order to forward requests to our *lister*, we need to add it to the *pattener*:

```
patterner.add('', lister)
```

Now our *lister* is added to the *pattener* and it has the home page prefix (that is empty string). Now open your browser to *localhost:8000*.

Wait a second!! We have the ugly no icon page again.. Furthermore, it only lists two prefixes: *video/* and *music/*. What about *__icons__*?!

For the ugly page without icons, it is OK because that is what we will fix in the next section.

For the *__icons__/* prefix, it is not listed because the *PatterLister* by default hides any prefix that starts and ends with two underscores. *__icons__/* does start and ends with two underscores and that is why it is not listed. We can change that if we wanted. However, we should leave the icons prefix unlisted because this prefix is only used for icons and is not intended to be opened by users directly.

2.6.2 Icons for our prefixes

Now we want to get rid of the no-icon ugliness. To do that, we first need to make a zip file which contains icons for each prefix. So lets make a zip file that contains the following:

- An image named *video.png*.
- An image named *music.png*.

The image names must be the same as the prefix after removing all trailing slashes. The file format we used above is *png*. You could use any other format if you like as long as the file name without extension is the same as the prefix.

Second, we need to tell our *lister* to get the icons from this zip file. Save the zip file in the same directory as our python file and name it *icons.zip*. Now, we modify our *PatternLister* object creation to the following:

```
lister = PatternLister(patterner, root='icons.zip')
```

Now our lister shows the icons we have put in the zip file.

Now our latest program is:

```
from cofan import *

from cofan import *

patterner = Patterner()
iconer = Iconer()
video = Filer('/home/user/Videos/', iconer=iconer)
music = Filer('/home/user/Music/', iconer=iconer)
#this is our lister
lister = PatternLister(patterner, root='icons.zip')

#add pages prefixes
patterner.add('__icons__/', iconer)
patterner.add('video/', video)
patterner.add('music/', music)

#remember: the home page prefix is always added last
patterner.add('', lister)

#make the handler use our patterner as its provider
handler = BaseHandler(patterner)

server = cofan.Server(('localhost', 8000), handler)

server.serve_forever()
```

2.6.3 Adding title to our pages

If you open the *lister* page, you will notice that the title shown in our *lister* for each page is the same as the page prefix but without a trailing slash. That is, our *video/* page is called *video* in the *lister*. The *music/* page is called *music* in the *lister*. May be you want to change that. For example, you want to start the title with a capital letter like *Music*. Or may be your users are Russian and you want the title to be . You can do that by giving the title as an argument to *Patterner.add()*:

```
patterner.add('music/', music, '')
```

Now we have changed the title for the *music/* page to be . You can change it to anything else.

2.6.4 Next

In the next lesson, we will serve files which are compressed in a zip file.

2.7 Serving Zip Files

In the previous lesson, we made a home page that lists our music page video page. In this lesson, we will add a little more thing. We will serve files but not from a directory. We will serve files from within a zip file.

2.7.1 Ziper

The *Zipper* provider is very similar to the *Filer*. But instead of taking a directory or file to serve, it takes a zip file. When a request comes to the *Ziper*, it searches its zip file for the requested path. If found, it uncompresses the file and sends it to the client.

Assume we want to serve the zip file in */home/user/myarchive.zip*. The first thing to do is create a *Ziper* and tell it where our zip file is. We also need to tell it what iconer to use. We will use the same iconer we used for the video filer and the music filer:

```
ziper = Ziper('/home/user/myarchive.zip', iconer=iconer)
```

Second, we give our ziper a prefix and add it to the *pattener* just like what we did to the other providers:

```
patterner.add('zip/', ziper)
```

Our program now becomes:

```
from cofan import *

patterner = Patterner()
iconer = Iconer()
video = Filer('/home/user/Videos/', iconer=iconer)
music = Filer('/home/user/Music/', iconer=iconer)
#this is our lister
lister = PatternLister(patterner, root='icons.zip')
ziper = Ziper('/home/user/myarchive.zip', iconer=iconer)

#add pages prefixes
patterner.add('__icons__/', iconer)
patterner.add('video/', video)
```

(continues on next page)

(continued from previous page)

```
patterner.add('music/', music)
patterner.add('zip/', ziper)

#remember: the home page prefix is always added last
patterner.add('', lister)

#make the handler use our patterner as its provider
handler = BaseHandler(patterner)

server = cofan.Server(('localhost', 8000), handler)

server.serve_forever()
```

Now open your browser to *localhost:8000*. You will see that our *ziper* is added to the home page because we added it to the *patterner*. Because we did not put an icon for it in the *lister* icons zip file, you will find the *ziper* has no icon. You can add an icon for it if you want.

If you click on the iconer link, you will see that the *ziper* behaves just like our *filer*. You can browse the files in the zip archive and download any of them.

2.7.2 Serving a static website

There are times when you want to serve a collection of html files as a website. Usually, static html files website is just a directory or a zip file with html files inside it. The home html file is called *index.html*.

When a *Filer* or a *Ziper* receives a request that points to a directory and that directory contains a file named *index.html*, the *Filer* or *Ziper* will not list the directory content. Instead, it will consider the directory to be a static html website and will redirect the request to the *index.html* file.

I will leave the practice to test this for you to do.

2.7.3 Next

We have gone through the main features of *cofan*. In the next lesson, we will go through a quick overview of other *cofan* features.

2.8 Other Features

We have highlighted the main features of *cofan*. In this lesson, we will have a quick look on the other classes in the library.

2.8.1 Statuser

The *Statuser* provider is used to respond to requests with a constant status code. Below is an example of how to create a *Statuser* object which always replies with *404 Not Found* status code:

```
not_found = Statuser(http.HTTPStatus.NOT_FOUND)
```

2.8.2 IPPatterner

The *IPPatterner* provider is used to respond differently based on the ip client address. It is similar to the *Patterner* but it matches the beginning of the client ip address instead of the url. The prefix is a regular expression string. Below is an example of how to create an *IPPatterner* object which always forwards requests to a filer only if the IP address of the client is in the local network (that is 192.168.1.xxx):

```
filer = Filer('/home/user/Videos/')
ippatterner = IPPatterner()
#add a pattern
ippatterner.add('192[.]168[.]1[.]', filer)
```

If the client address does not start with any added regular expression, the client gets a *401 UNAUTHORIZED* response.

2.8.3 JFiler

The *JFiler* provider is the same as the *Filer* provider except it serves directory content as a JSON object and it does not redirect to *index.html* even if it is in a requested directory. Obviously it does not take any iconer. Below is an example of creating a *JFiler*:

```
jfiler = JFiler('/home/user/Videos/')
```

2.8.4 Next

This is everything in the *cofan* tutorial. You can have a look at *cofan* reference at any time if you need detailed information.

2.9 cofan Reference

Date 2019-01-30

Version 0.0.3

Authors

- Mohammad Alghafli <thebsom@gmail.com>

cofan is an http server library for serving files and any other things. You use it to share content in the form of a website. The current classes give you the following:

- Serve the content of a local directory in a form similar to file browser with icons for directories and files based on their extension.
- Serve the content of a local zip file the same way as the local directories.
- List the content of a local directory in json form.
- Serve local html files as a web site.
- Organize your urls in prefix trees.
- Response differently for different ip addresses

It also supports requests of partial files to resume previously interrupted download.

Here is an example of how to use it:

```
from cofan import *
import pathlib

#our site will be like this:
#   /           (this is our root. will list all the branches below)
#   |
#   |- vid/
#   |  this branch is a file browser for videos
#   |
#   -- site/
#      this will be a web site. just a collection of html files

#lets make an http file browser and share our videos
#first, we specify the icons used in the file browser
#you can omit the theme. it defaults to `humanity`
icons = Iconer(theme='humanity')

#now we create a Filer and specify the path we want to serve
vid = Filer(pathlib.Path.home() / 'Videos', iconer=icons)

#we also want to serve a web site. lets create another filer. since the root
#directory of the site contains `index.html` file, the filer
#will automatically redirect to it instead of showing a file browser
#no file browser also means we do not need to specify `iconer`
#parameter. you can still use it if you want but that would not be very
#useful
site = Filer(pathlib.Path.home() / 'mysite')

#now we need to give prefixes to our branches
#we create a patterner
patterns = Patterner()

#then we add the iconer, filers with their prefixes
#make sure to add a trailing slash
patterns.add('vid/', vid)
patterns.add('site/', site)
#we also need to add our iconer
#you need to tell the iconer about its prefix. by default it assumes
#`__icons__`
patterns.add('__icons__/', icons)

#now we have all branches. but what if the user types our root url?
#the path we will get will be an empty string which is not a prefix of any
#branch. that will be a 404
#lets make the root list and other branches added to `patterns`
#the branches will be shown like the file browser but now the icons will be
#for the patterns instead of file extensions
#we need to specify where the icons are taken from
#the icons file should contain an icon named `vid.<ext>` and an icon named
#`site.<ext>` where <ext> can be any extension.
root = PatternLister(patterns, root=pathlib.Path.home() / 'icons.zip')

#and we add our root to the patterns with empty prefix
patterns.add('', root)

#now we create our handler like in http.server. we give it our patterner
handler = BaseHandler(patterns)
```

(continues on next page)

(continued from previous page)

```
#and create our server like in http.server
srv = Server(('localhost', 8000), handler)

#and serve forever
srv.serve_forever()

#now try to open your browser on http://localhost:8000/
```

This module can also be run as a main python script to serve files from a directory.

commandline syntax:

```
python -m cofan.py [-a <addr>] [<root>]
```

options:

- -a <addr>, –addr <addr>: specify the address and port to bind to. <addr> should be in the form *<ip>:<port>* where *<ip>* is the ip address and *<port>* is the tcp port. defaults to *localhost:8000*.

args:

- root: The root directory to serve. Defaults to the current directory.

class cofan.**BaseHandler**(*provider*)

Base http handler class in cofan library. It holds a provider instance and gets the response of all requests from it. Note that unlike what is usually done in http.server, when creating an http.HTTPServer instance, you should pass an instance of *BaseHandler* instead of the class itself. For example:

```
myprovider = Filer('/path/to/my/directory')
myhandler = BaseHandler(myprovider)
srv = http.server.HTTPServer(('localhost', 80), myhandler)
```

BaseHandler has one class attribute:

__header_modifiers__: A tuple of request headers which have modifier methods in this class. For each header present in a request, *self.mod_<header>()* is called where <header> is the header name. The modifier method must return the response, headers and content after any necessary modification. It must take the following arguments:

- response: Last response status code after any previous modifications by other modifier mehtods.
- headers: Last headers after any previous modifications by other modifier mehtods.
- content: Last content file after any previous modifications by other modifier mehtods.

This tuple currently has only one header: *Range*. See *self.mod_Range()* for the modification applied.

do_GET()

Calls *BaseProvider.get_content()* from *self.provider* and sends the returned response, headers and content. After this process is done, the content file is closed and the returned post processing fucntion is called. This happens regardless of whether the process ended with success or not.

do_HEAD()

Same as *self.do_GET()* but does not send any content.

get_response()

Called by *self.do_GET()* and *self.do_HEAD()*. Parses the url path and query string and replaces all url escapes. After that, it gets the response *self.provider.get_content()*. Before returning the response, headers, content file and post processing callables, it calls any modifier method that should be called. If the method

fails while applying modifications, it closes the content file and calls the post processing callable before propagating the exception.

mod_Range(*response*, *headers*, *content*)
Called when the *Range* header is present in the request. If the response status code is 200 (OK) and the content file is seekable, the status code is changed to 206 (Partial Content) and the content file is changed to a partial file pointing to the requested range. Response status code is changed to 416 (Requested Range Not Satisfiable) if the range start is not between 0 and total size or if the range end is not between start and total size.

class cofan.**BaseProvider**
basic provider class. each time a request is intended to this provider, the provider should return a response code, headers and body. it defines two methods:

- get_content(): this should return the response code, headers and body
- short_response(): helper method to send a response code and its description.

You probably want to use one of *BaseProvider* subclasses or inherit it in your own class.

classmethod get_content(*handler*, *url*, *query={}*, *full_url=''*)
This method is called by the *http.server.BaseHTTPRequestHandler* object when a request arrives. It must returns a tuple containing:

- Response code
- Response headers
- Response body
- Postprocessing callable

The response code should be an integer or a member of http.HTTPStatus. Response headers should be a dictionary where keys are header keywords and values are header values. The body must be a readable file like object. Postprocessing callable is a callable object which takes no arguments. It is called after the request is served. The body object is closed automatically after serving the request and does not need to be closed in the postprocessing callable.

This method should be overridden in subclasses. The default implementation is to send OK response code with a body that contains short description of the response code.

args:

- handler (*http.server.BaseHTTPRequestHandler*): The object that called this method.
- url (str): The url that was requested after removing all prefixes by other providers. Look at *Patterner* for information of how prefixes are stripped.
- query (dict): Request query arguments. Defaults to empty dictionary.
- full_url (str): The full url that was requested without removing prefixes. Defaults to empty string

returns:

- response (http.HTTPStatus): Response code.
- headers (dict): Response headers.
- content (binary file-like object): The response content.
- postproc (callable): A callable that will be called after serving the content. This callable will be called as long as the *get_content()* method succeded regardless whether sending the content to the client succeded or not. The intention of this callable is to close all files other than *content* in case there are open files. For example, if *content* is a file inside a zip file, closing *content* is not enough without closing the parent zip file.

static short_response(*response=<HTTPStatus.OK: 200>*, *body=None*)

Convinience method which can be used to send a status code and its description. It returns the same values as *get_content* but the code is specified as a parameter and the body defaults to a description of the code.

args:

- response (http.HTTPStatus): Response code to send. Defaults to *http.HTTPStatus.OK*.
- body: The value to send in the body. It defaults to *None* which sends a short description of the code. If the body is not a bytes object, it is converted to a string and then encoded to utf8.

returns:

- response (http.HTTPStatus): Response code given in the args.
- headers (dict): Response headers dict with *text/html* in *Content-Type* header and the length of the body in *Content-Length* header.
- content (binary file-like object): Description of *response* as a utf-8 encoded string if *body* arg is *None*. Otherwise, returns *body* represented as utf-8 encoded string.
- postproc: Always a value of *None*.

class cofan.**Filer**(*root*, *iconer=None*)

Same as *JFiler* when the url points to a file. If the url points to a directory, the content of the directory is sent as an html file presenting directory content as a file browser with icons instead of JSON.

get_content(*handler*, *url*, *query={}*, *full_url=''*)

Serves files based on the url given starting from the *self.root*. If the url points to a file, the file is served. *Last-Modified* header is sent to help clients cache the file. If the url points to a directory, a list of files it contains is served as a JSON object with 2 members:

- dirs: List of directories under the requested directory. If *full_url* parameter is not an empty string, a '..' value is added to the dirs list.
- files: List of files under the requested directory.

If the url does not point to a file or directory under *self.root*, it responds with NOT FOUND.

serve_dir(*handler*, *url*, *query={}*, *full_url=''*)

Overrides *JFiler.serve_dir()*. Sends an html file containing subdirectories and files present in the requested directory in a similar way to file browsers. Icons are taken from *self.iconer*.

class cofan.**IPPatterner**

The same as *Patterner* but relays requests based on IP address patterns instead of url.

get_content(*handler*, *url*, *query={}*, *full_url=''*)

Searches the pattern list for a pattern that matches the beginning of the url and relays the request to the corresponding provider. If not found, sends a *NOT FOUND* response. The search is done from the first added pattern. If two patterns overlap, the more specific pattern should be added first. For example, *files/mysite/* should be added before *files/*. Otherwise, any request to *files/mysite/* will be served by *files/* since it will be checked first.

class cofan.**Iconer**(*root=None*, *theme='humanity'*, *prefix='__icons__/'*)

Same as *Ziper* but used to serve file icons used in *Filer* and *Ziper*. Defines methods to help other objects find icon urls for files of different types. The root of *Iconer* is a zip file which contains image files in its toplevel. The name of each image file should be in the form <*name*>.<*ext*> where <*ext*> can be any extension and <name> can be any of the following:

- The string *directory*. It makes the image used as the icon for directories.
- A file extension. It makes the image used for file of this extension.

- General mimetype (such as audio, video, text, ...). It makes the image used for this mimetype if the file extension image does not exist.
- The string *generic*. It makes the image used as a fallback icon if none of the above was found.

Any file without extension in *self.root* is ignored.

get_content(*handler*, *url*, *query={}*, *full_url=''*)
Overrides *Filer.get_content()*. Gives the same result as *Filer.get_content()* but looks at the content of a zip file instead of a directory.

get_icon(*name*)
This method is to be used in other content providers to get icons. Returns the url of an icon for *name*. The icon is constructed in the following way:

- The extension is extracted from *name*. If *name* has no extension, the full value of *name* is taken.
- If there is a file in *self.root* named as the extension of *name* extracted in the previous step, the url of this file is returned. For example, if *name* is *foo.mp4*, this method will look for a file named *mp4.<extension>* where *<extension>* may be any string. The extension is case insensitive so *foo.mp4* will be the same as *foo.MP4*.
- If there is no file found in the previous step, the mimetype is guessed and the general type is taken (such as audio, video, etc...).
- If there is a file in *self.root* with the same name as the generel mimetype extracted in the previous step, the url of this file is returned. For example, if *name* is *foo.mp4*, this method will look for a file named *mp4.<anything>* first. If it there is no such file in *self.root*, the method will look for *video.<anything>*. *<anything>* may be any string.
- If there is no file found in the previous step, this method looks for a file named *generic.<anything>* and the url of this file is returned.
- If there is no file found in the previous step, an empty string is returned.

get_icons()
Used in *self.__init__()*. Looks at the toplevel content of *self.root* for any files and makes a dictionary for each file. The keys of the dictionary are file names without extensions and the values are the file names with extensions. Any files without extension are ignored. The dictionary is used to look for icons without opening the zip file.

class cofan.**JFiler**(*root*)
Lists directory contents and serves files from local file system based on the recieved url.

exists(*url*)
Returns *True* if the url points to an existing file or directory under *self.root*. Otherwise returns *False*.

args:

- url (path-like object): The recieved url to check.

returns: True if *url* is an existing file or subdirectory in *self.root*. False otherwise.

get_content(*handler*, *url*, *query={}*, *full_url=''*)
Serves files based on the url given starting from the *self.root*. If the url points to a file, the file is served. *Last-Modified* header is sent to help clients cache the file. If the url points to a directory, a list of files it contains is served as a JSON object with 2 members:

- dirs: List of directories under the requested directory. If *full_url* parameter is not an empty string, a '..' value is added to the dirs list.
- files: List of files under the requested directory.

If the url does not point to a file or directory under *self.root*, it responds with NOT FOUND.

get_file_list(*url*, *parent=False*)
Called by *self.get_content*. Returns a dictionary containing 2 members:

- dirs: A list of directories under the directory pointed by url.
- files: A list of files under the directory pointed by url.

args:

- url (path-like object): The directory url to list its content.
- parent (bool): If *True*, adds '..' to the dirs list. Defaults to False.

is_dir(*url*)
Returns *True* if the url points to an existing directory under *self.root*. Otherwise returns *False*.

args:

- url (path-like object): The recieved url to check.

return: True if *url* is an existing subdirectory of *self.root*. False otherwise.

is_file(*url*)
Returns *True* if the url points to an existing file under *self.root*. Otherwise returns *False*.

args:

- url (path-like object): The recieved url to check.

returns: True if *url* is an existing file in *self.root* or one of its subdirectories. False otherwise.

serve_dir(*handler*, *url*, *query={}*, *full_url=''*)
Called by *self.get_content()* when the url points to a direcotry. Gets a list of directories and files under the requested directory pointed by *url* argument.

Takes the same arguments as *self.get_content()*.

serve_file(*handler*, *url*, *query={}*, *full_url=''*)
Called by *self.get_content()* when the url points to a file. Serves the file content as a JSON object.

Takes the same arguments as *self.get_content()*.

class cofan.**PatternLister**(*provider*, *root=None*, *exclude='|__.*__/?'*, *include='.*'*)
Similar to *Filer* but instead of showing content of a directory, shows the prefixes added to a *Patterner*. It also provides icon urls for the prefixes. See *Patterner* for more details of the *Patterner* provider.

get_content(*handler*, *url*, *query={}*, *full_url=''*)
Overrides *Iconer.get_content()*. If the url is empty string, returns a list of prefixes in *self.provider*. If the url starts with *__pattern_icons__/*, returns an icon from *self.root*.

get_icon(*name*, *full_url=''*)
Returns the icon of the prefix *name*. Unlike *Iconer*, this method takes *name* itself after striping any trailing slashes as the name of the icon.

serve_patterns(*handler*, *url*, *query={}*, *full_url=''*)
Same as *Filer.serve_dir()* but returns prefixes from *self.provider* instead of directories. Used in *self.get_content()*.

class cofan.**Patterner**
BaseProvider subclass that relays requests to other providers based on requested url pattern. The other providers are added to the *Patterner* instance with the request url pattern they should get. When the *Patterner* gets

a request, it searches for a pattern that matches the beginning of the url. When found, the *Patterner* calls *get_content()* method of the target provider, giving it the same parameters but with the prefix stripped from the beginning of the url.

add(*pattern*, *provider*, *title=''*)
Adds a pattern and a provider to the *Patterner*.

args:

- pattern (str): A string containing the url prefix of the provider.
- provider (*BaseProvider*): The provider to relay the request to.

get_content(*handler*, *url*, *query={}*, *full_url=''*)
Searches the pattern list for a pattern that matches the beginning of the url and relays the request to the corresponding provider. If not found, sends a *NOT FOUND* response. The search is done from the first added pattern. If two patterns overlap, the more specific pattern should be added first. For example, *files/mysite/* should be added before *files/*. Otherwise, any request to *files/mysite/* will be served by *files/* since it will be checked first.

get_patterns()
Returns the pattern strings added to the *Patterner*.

get_title(*pattern*)
Returns the pattern title.

args:

- pattern (str): the pattern to look for its title.

returns:

The pattern title.

remove(*pattern*)
Removes a pattern and its provider from the pattern list. If a pattern exists multiple times (which you should not do anyway), only the first occurance is removed.

args: pattern (str): Pattern to remove.

class cofan.**Server**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
same as *http.server.HTTPServer* but handles requests in daemon threads.

class cofan.**Statuser**(*response=<HTTPStatus.NOT_FOUND: 404>*)
A subclass of *BaseProvider*. This provider is very similar to BaseProvider. The only difference is that it takes the response code in its constructor and sends this response code instead of OK.

get_content(*handler*, *url*, *query={}*, *full_url=''*)
This method is called by the *http.server.BaseHTTPRequestHandler* object when a request arrives. It must returns a tuple containing:

- Response code
- Response headers
- Response body
- Postprocessing callable

The response code should be an integer or a member of http.HTTPStatus. Response headers should be a dictionary where keys are header keywords and values are header values. The body must be a readable file like object. Postprocessing callable is a callable object which takes no arguments. It is called after the request is served. The body object is closed automatically after serving the request and does not need to be closed in the postprocessing callable.

This method should be overridden in subclasses. The default implementation is to send OK response code with a body that contains short description of the response code.

args:

- handler (*http.server.BaseHTTPRequestHandler*): The object that called this method.
- url (str): The url that was requested after removing all prefixes by other providers. Look at *Patterner* for information of how prefixes are stripped.
- query (dict): Request query arguments. Defaults to empty dictionary.
- full_url (str): The full url that was requested without removing prefixes. Defaults to empty string

returns:

- response (http.HTTPStatus): Response code.
- headers (dict): Response headers.
- content (binary file-like object): The response content.
- postproc (callable): A callable that will be called after serving the content. This callable will be called as long as the *get_content()* method succeded regardless whether sending the content to the client succeded or not. The intention of this callable is to close all files other than *content* in case there are open files. For example, if *content* is a file inside a zip file, closing *content* is not enough without closing the parent zip file.

class cofan.**Ziper**(*root*, *iconer=None*)

Same as *Filer* but serves the content of a zip file instead of a directory. The root in the constructor must be a zip file. Files served from this class are not seekable so resuming download is not possible for the content of a Ziper.

exists(*url*, *root=None*)

Overrides *Filer.exists()*. Looks at the content of the zip file instead of a directory.

get_content(*handler*, *url*, *query={}*, *full_url=''*)

Overrides *Filer.get_content()*. Gives the same result as *Filer.get_content()* but looks at the content of a zip file instead of a directory.

get_file_list(*url*, *root=None*, *parent=False*)

Overrides *Filer.get_file_list()*. Looks at the content of the zip file instead of a directory.

is_dir(*url*, *root=None*)

Overrides *Filer.is_dir()*. Looks at the content of the zip file instead of a directory.

is_file(*url*, *root=None*)

Overrides *Filer.is_file()*. Looks at the content of the zip file instead of a directory.

serve_dir(*handler*, *url*, *query={}*, *full_url=''*, *root=None*)

Overrides *Filer.serve_dir()*. Looks at the content of a zip file instead of a directory.

This method has an additional optional argument:

root (*zipfile.ZipFile*): Opened *self.root* It is used in *self.get_content()* to avoid opening and closing *self.root* multiple times. Defaults to None which means *self.root* will be opened and a new *zipfile.ZipFile* instance will be created.

serve_file(*handler*, *url*, *query={}*, *full_url=''*, *root=None*)

Overrides *JFiler.serve_dir()*. Looks at the content of the zip file instead of a directory.

This method has an additional optional argument:

root(*zipfile.ZipFile*) Opened *self.root*. It is used in *self.get_content()* to avoid opening and closing *self.root* multiple times. Defaults to None which means *self.root* will be opened and a new *zipfile.ZipFile* instance will be created.

2.10 Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index